C4

```
/***********************************************************************
                                                                  ******
**
** File Name:   RSLstart.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**        The intent of the contents of this file is to implement the
**        functions the control execution of the restore for the
                                                        Restore Service
**        Library.
**
**        These functions are provided to allow:
**       - creation of submit objects,
                                   which define the set of objects to be
**         restored and the scripts to be run before and after
                                                           restoration,
**       - starting the restoral of a submit object.
**
** The following functions comprise restoral management:
**
**       RSTSL_Start
**
**
** Compile-Time Options:
**               This section must list any compile time definitions
**               which will affect this header.
**
***********************************************************************
                                                                 *****/


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/wait.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
```

```
#include <RSLinterns.h>
#include <RSLauxSupp.h>
#include <restore/EDMRESubmitApi.h>

#include <restore/REprogmsg.h>
#include <restore/dispatch_daemon.h>
#include <restore/EDMREProgressApi.h>

extern int RunExecutable(const boolean_ty ResetUid,
                         const int RunUid,
                         const char *starting_cwd,
                         const char *executable_name,
                         char **executable_argv,
                         char **executable_env,
                         int *run_exit_status,
                         boolean_ty *run_cancelled,
                         boolean_ty (*QuitTest)(void));

extern int RunWorkItemRestores(int, boolean_ty (*CancelRestoreTest)());

static eerrno_ty
ExecuteWorkItemRestore(int SubmitObjectID,
                       boolean_ty (*QuitTest)(void));

static eerrno_ty
RunPrepareRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int *PrepareExit);
static eerrno_ty
RunCleanupRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int runphase_status,
                  int *CleanupExit);

/*
 * #defines, structures, typedefs local to this source file
 */

#define STR_SURE(str) (str) ? str:""
#define REMOVE_NEWLINE(str)\
{\
   int rem_nl_index;\
   for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
     {\
       if(str[rem_nl_index] == '\n')\
          str[rem_nl_index] = '\0';\
     }\
}

/***********************************************************************
 * Start
 *
 * This function begins execution of the restoral of the objects in a
 * submit object.  Its progress and requests for operator input are
 * returned via RSTSL_GetRestoreFeedback.
 *
 * Parameters:
 *
 * SubmitObjectID (
              I) - ID of the submit object which describes the restore
 * QuitTest        (I) - function to call to check for quit signal
 *
 **********************************************************************/
eerrno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest)(
                                                                  void))
{
 int ret_pre;
 int ret_exec;
 int ret_post;
 int ret_all_ok;
 int PrepareExit = 0;
 int CleanupExit = 0;
 boolean_ty QuitFlag = FALSE;

 char asctime[32];

 memset(asctime, 0, 32);

 (void)time(&rcp->rc_cmd_starttime);

 (void)ctime_r(&rcp->rc_cmd_starttime, asctime, 32);

 rcp->rc_cmd_last_waf_time = rcp->rc_cmd_starttime;

 REMOVE_NEWLINE(asctime);

 rbe_log_stats(0, "Restore Started at %s.", asctime);

 rbe_log_stats(0, "Restore Started application type %s.",
                 (rcp->rc_backup_app == 0)
                         ? "Network"
                         : ((struct pluginIDdata *)(
                                   rcp-> currentPIptr-> idData))
                                                             ->
                                                         name );

 rbe_log_stats(0, "Restore Started of "\
                 "top level object: %s. template %s, trailset %s.",
                 STR_SURE(rcp->rc_top_level_object_name),
                 STR_SURE(rcp->rc_template_name),
                 (rcp->rc_saveset_thread) ? "Alternate": "Primary");


 rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
                 STR_SURE(rcp->rc_human_uidname),
                 rcp->rc_human_uid, rcp->rc_human_gids[0]);

 rbe_log_stats(0, "Restore Started with client destination %s.",
                 STR_SURE(rcp->rc_client_hostname));

 /* if not a network restore,
                         check if plugin has its own start function */

 if ( rcp-> rc_backup_app != 0
   && NULL != rcp-> currentPIptr-> piFuncArray[
                                      PIFuncIndexStartRestore ] )
 {
   setGlobalStatus( EDMRE_STATE_EXECUTE );
                                      /* set RE's internal status */
   ret_exec = rcp-> currentPIptr-> piFuncArray[
                                        PIFuncIndexStartRestore ]
                                                 (
                                  rcp, SubmitObjectID, QuitTest);

   if( QuitTest() )      /* check for abort before return */
   {
     rbe_log_stats( EP_RB_RECOVER_ABORT,
                    "The restore was quit by the user during
                                                    execution.");
     setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                      /* set RE's internal status */
```

```
       return EP_RB_RECOVER_ABORT;
     }

     if (E_SUCCESS != ret_exec)
       setGlobalStatus( EDMRE_STATE_FAILED );
                                         /* set RE's internal status */
     else
       setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                         /* set RE's internal status */

     return( ret_exec );
   }

   ret_pre = RunPrepareRestore(SubmitObjectID,
                               QuitTest,
                               &PrepareExit);

   if(EP_RB_RECOVER_ABORT == ret_pre)
   {
     rbe_log_stats(EP_RB_RECOVER_ABORT,
                   "The restore was quit by the user during
                                                  preparation.");
     setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                         /* set RE's internal status */
     return EP_RB_RECOVER_ABORT;
   }
   if(PrepareExit != 0)
   {
     rbe_log_stats(EP_RB_RECOVER_PREFAILED,
                   "The restore failed during preparation. Exit %d",
                   PrepareExit);

     setGlobalStatus( EDMRE_STATE_FAILED );
                                         /* set RE's internal status */
     return (EP_RB_RECOVER_PREFAILED);
   }

   setGlobalStatus( EDMRE_STATE_EXECUTE );
                                         /* set RE's internal status */

   ret_exec = ExecuteWorkItemRestore(SubmitObjectID,
                                     QuitTest);

   QuitFlag = QuitTest();        /* check for abort before cleanup */


   if (E_SUCCESS == (
           ret_all_ok = ret_exec) )     /* check if any WIs failed */
   {
     int local_stat;
     EDMStats stats;
     memset( &stats, 0, sizeof(EDMStats) );

     if (0 != getRestoreStatus( 0, &stats, &local_stat ))
     {
       rbe_log_stats(
           local_stat, "Internal error: Failed in getRestoreStatus.");
       ret_exec = ret_all_ok = EP_RB_RECOVER_EXECUTEFAILED;
     }
     else if (stats.edm.failed)
     {     /* if any workitems failed,
                                its a failure for cleanup purposes */
         if (0 == stats.edm.successful)
           ret_all_ok = EP_RB_RECOVER_ALLFAIL;
         else if (stats.edm.successful > stats.edm.failed)
           ret_all_ok = EP_RB_RECOVER_FEWFAIL;
```

```
          else
            ret_all_ok = EP_RB_RECOVER_MANYFAIL;
      }
  }

  ret_post = RunCleanupRestore(SubmitObjectID,
                               QuitTest,
                               ret_all_ok,
                               &CleanupExit);

  if(QuitFlag)
  {
    rbe_log_stats(EP_RB_RECOVER_ABORT,
                  "The restore was quit by the user during execution.
                                                                       ");
    setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                           /* set RE's internal status */
    return EP_RB_RECOVER_ABORT;
  }

  if (E_SUCCESS != ret_exec)      /* return execute status if it failed
                                                                      */
  {
    setGlobalStatus( EDMRE_STATE_FAILED );
                                           /* set RE's internal status */
    return ret_exec;
  }

  if(EP_RB_RECOVER_ABORT == ret_post)
  {
    rbe_log_stats(EP_RB_RECOVER_ABORT,
                  "The restore was quit by the user during cleanup.");
    setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                           /* set RE's internal status */
    return EP_RB_RECOVER_ABORT;
  }

  if( (CleanupExit != 0) || (E_SUCCESS != ret_post) )
  {
    rbe_log_stats(EP_RB_RECOVER_POSTFAILED,
                  "The restore failed during cleanup. Exit %d",
                  CleanupExit);

    setGlobalStatus( EDMRE_STATE_FAILED );
                                           /* set RE's internal status */
    return (EP_RB_RECOVER_POSTFAILED);

  }

  setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                           /* set RE's internal status */

  return( E_SUCCESS );

}        /* RSTSL_Start */
```

```
static eerrno_ty
ExecuteWorkItemRestore(int SubmitObjectID,
                        boolean_ty (*QuitTest)(void))


{
    int ret_RunWItem;
    sm_push();

    rcp->error_message[0] = 0;

    if(0 != (ret_RunWItem = RunWorkItemRestores(
                                SubmitObjectID, QuitTest)))
    {
      rbe_log_stats(0,"Internal error: Failed in RunWorkItemRestores");
    }

    sm_pop();

    if (QuitTest() == TRUE)
        return EP_RB_RECOVER_ABORT;

    if (ret_RunWItem != 0)
        return EP_RB_RECOVER_EXECUTEFAILED;

    return E_SUCCESS;
}
```

```
#define EXECUTABLE_MAX 1024
static eerrno_ty
RunPrepareRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int *PrepareExit)
{
  char **prephaseargs = NULL;
  char **prephaseenv = NULL;
  int GetSOstatus = 0;
  char preExecutable[EXECUTABLE_MAX];
  boolean_ty restore_cancelled = FALSE;

  *PrepareExit = 0;

  /*
   * GetSOPrePhase allocates prephaseargs & prephaseenv.
   * This will need to be free'ed later.
   *
   */

  if(0 != GetSOPrePhase(SubmitObjectID,
                        preExecutable,
                        EXECUTABLE_MAX,
                        &prephaseargs,
                        &prephaseenv,
                        &GetSOstatus))
  {
    rbe_log_stats(0,"Internal error: Failed in GetSOPrePhase");
    return (EP_RB_RECOVER_FATALERR);
  }

  if(0 != strcmp(preExecutable, ""))
  {
    setGlobalStatus( EDMRE_STATE_PREPHASE );
    if(-1 == RunExecutable(FALSE,
                           0,
                           NULL,
                           preExecutable,
                           prephaseargs,
                           prephaseenv,
                           PrepareExit,
                           &restore_cancelled,
                           QuitTest))
    {
      rbe_log_stats(
            0,"Internal error: Failed in RunExecutable for prepare.");
      return (EP_RB_RECOVER_FATALERR);
    }
    if(TRUE == restore_cancelled)
      return (EP_RB_RECOVER_ABORT);

  }

  return( E_SUCCESS );
}
```

```
boolean_ty alwaysFalse() { return FALSE; }
```

```
static eerrno_ty
RunCleanupRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int runphase_status,
                  int *CleanupExit)
{

  char **postphaseargs = NULL;
  char **postphaseenv = NULL;
  int GetSOstatus = 0;
  char postExecutable[EXECUTABLE_MAX];
  boolean_ty restore_cancelled = FALSE;
  boolean_ty ignore_quit=FALSE;

  *CleanupExit = 0;

  /*
   * GetSOPostPhase allocates postphaseargs & postphaseenv.
   * This will need to be free'ed later.
   *
   */
  PurgeTrailQueue();
  if(0 != GetSOPostPhase(SubmitObjectID,
                         postExecutable,
                         EXECUTABLE_MAX,
                         &postphaseargs,
                         &postphaseenv,
                         &GetSOstatus))
  {
    rbe_log_stats(0,"Internal error: Failed in GetSOPostPhase");
    return (EP_RB_RECOVER_FATALERR);
  }

#define RESTORE_BREAK       "RESTORE_BREAK="
#define RESTORE_BREAK_TRUE  "RESTORE_BREAK=T"
#define RESTORE_BREAK_ERROR "RESTORE_BREAK=E"

  if(0 != strcmp(postExecutable, ""))
   {
     /*
      * If a quit has been specified, we need to tweak the
      * RESTORE_BREAK environment variable if set
      */
     char *abort=NULL;
     if(QuitTest())
       {
          abort=RESTORE_BREAK_TRUE;
       }
     else if(0!=runphase_status)
       {
          abort=RESTORE_BREAK_ERROR;
       }

     /*
      * ignore_quit is set to 1 when we have already processed a BREAK
      * (CANCEL from gui), and are using the environment variable
      * RESTORE_BREAK_E to signal the post restore script to clean
      * up form this break.  When that happens, an ignore_quit value
      * of 1 will cause actual quit signals to be ignores by the
      * cleanup script, since we already know (via the environment
      * variable) that we are in "cleanup mode" and no further signal
      * intercpetion is necessary.
      */
     ignore_quit=FALSE;
     if(NULL!=abort && NULL!=postphaseenv)
```

```
       {
          int isub=0;
          char *cptr;
          while(cptr=postphaseenv[isub])
            {
              if(strncmp(postphaseenv[isub],RESTORE_BREAK,strlen(
                                                  RESTORE_BREAK))==0)
                {
                  postphaseenv[isub]=esl_strdup(abort);
                  ignore_quit=TRUE;
                  if(NULL==postphaseenv[isub])
                    {
                      rbe_log_stats(
              EP_RB_RECOVER_MALLOC_FAILURE,"Allocate failed in RSLstart.c");
                      return EP_RB_RECOVER_POSTFAILED;
                    }
                }
              isub++;
            }
       }

     setGlobalStatus( EDMRE_STATE_POSTPHASE );
                                              /* set RE's internal status */
     if(-1 == RunExecutable(FALSE,
                            0,
                            NULL,
                            postExecutable,
                            postphaseargs,
                            postphaseenv,
                            CleanupExit,
                            &restore_cancelled,
                            ignore_quit?alwaysFalse:QuitTest))
     {
       rbe_log_stats(
           0,"Internal error: Failed in RunExecutable for cleanup.");
       return (EP_RB_RECOVER_FATALERR);
     }
     if(TRUE == restore_cancelled)
       return (EP_RB_RECOVER_ABORT);

  }
  return( E_SUCCESS );
}
```

```
static eerrno_ty
RunExecutionOverrideRestore(int SubmitObjectID,
                            boolean_ty (*QuitTest)(void))
{

  return( E_SUCCESS );
}
```

```
#undef EXECUTABLE_MAX
```

```
/**********************************************************************
                                                                ******
**
** File Name:   RSLwisvr.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions the control execution of work item restores for
                                                              Restore
**      Service  Library.
**
**      These functions are provided to allow:
**
** The following functions comprise restoral management:
**
**      RunWorkItemRestores()
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
**********************************************************************
                                                                *****/

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <values.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <restore/REprogmsg.h>

/*
 * Local headers
 */

#include <RSLspexits.h>
#include <RSLremfd.h>
#include <EDMRESchedApi.h>
#include <EDMREHandleMgrApi.h>
#include <RSLinterns.h>
#include <RSLrbrmain.h>
#include <restore/EDMRESubmitApi.h>
#include <EDMREDrainApi.h>

#define STR_SURE(str) (str) ? str:""


/*
```

```
 *
 * RunWorkItemRestores
 * {
 * Set the number of drives being used for the life of the restore.
 * SetQuitFlag = FALSE
 * TrailRestoresLeft = { # of trail restores }
 * TrailRestoresRunning = 0;
 *
 *   while drive available.
 *     RunTrail -- Set drive concurrency for trail restore.
 *     TrailRestoresRunning++;
 *     while OKToRunWIForTrail
 *       StartWIRestore
 *
 *   while(1)
 *     if(QuitTest)
 *        Send WICancells
 *        SetQuitFlag = TRUE
 *     Select(from_pipe, timeout {5 seconds})
 *     for each WI that completes
 *       interperate return.
 *       Drain progress.
 *       Send final Progress for work item.
 *       if(WIfailed)
 *          if(OKtoReschedule && SetQuitFlag == FALSE)
 *             Add to TrailQueue
 *       if(TrailRestoreHasMoreWorkitems && SetQuitFlag == FALSE)
 *          while OKToRunWIForTrail
 *             StartWIRestore
 *       else -- RunNewTrailRestore
 *          EndTrailRestore(prevTrailQueue)
 *          TrailRestoresRunning--;
 *          TrailRestoresLeft--;
 *          while (drive available &&
 *                 SetQuitFlag == FALSE &&
 *                 TrailRestoresLeft)
 *
 *              RunTrail -- Set drive concurrency for trail restore.
 *              TrailRestoresRunning++;
 *              while (
                          OKToRunWIForTrail && SetQuitFlag == FALSE)
 *                StartWIRestore
 *              end while
 *          end while
 *       end else
 *    End for each WI completes
 *    if (((SetQuitFlag == TRUE) && (TrailRestoresRunning == 0))||
 *        TrailRestoresLeft == 0)
 *        return;  exit loop.
 * end while(1)
 *
 * }
 *
 */

/* Stubs */
static int
InterpretWorkItemRestoreResults(wi_restore_results *results);

static int
test_fd(int fd);

static void
DebugLogFds(char *error_msg,
            fd_set *fds);
```

```
static int
DetermineGlobalDriveUse();

static int
test_fd_hup(int fd);

static int
FindTrailIDForWItem(int handle,
                    int *TrailID,
                    int *status);

static int
SendRunningWorkItemsQuit();

/* End Stubs */

static int
Select(int nfds,
       fd_set *readfds,
       fd_set *writefds,
       fd_set *exceptfds,
       struct timeval *timeout);


static int
HandleWorkItemRestoreResults(int FromFD,
                             int *TrailID,
                             wi_restore_results *results);
static int
RunWorkItemRestoresForTrail(const int TrailID,
                            const int CountDrivesAvailable,
                            boolean_ty (*CancelRestoreTest)(),
                            boolean_ty *QuitFlag,
                            int *CountDrivesInUse);

/*
 * RunWorkItemRestores()
 *
 *  Runs a set of work item restores.
 *
 * Args:
 *  SubmitObject
 *  CancelRestoreTest().
 *
 * Returns: int 0 for success.
 */
int
RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)(
                                                                     ))
{
  boolean_ty QuitFlag = FALSE;  /* Has the user requested a quit.*/
  boolean_ty SentQuit = FALSE;  /* Have we initiated the quit. */

  int TrailRestoresRunning = 0;
                          /* The number of trail restores running. */
  int TrailRestoresLeft;
                             /* The number of trail restores left. */
  int TrailRestoresTotal;
                            /* The number of trail restores total. */

  int CountDrivesAvailable;      /* The count of drives available. */
  int CountDrivesInUse = 0;      /* The count of drives in use. */

  int temp_status;
  int HighestActiveTrail = 0;
                         /* The trail queues are ordered from 1 to n. */
```

```
                            /* This is the highest trail running
                                                                */

  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: Running RunWorkItemRestores.");
  }
  /* GenerateTrailQueues()
   * Buckets the work items into trail queues.
                                          The trail queues are sorted
   * in the order which the restores should run.
   */

  if(0 != GenerateTrailQueues(SubmitObject,
                              &TrailRestoresTotal,
                              &temp_status))
  {
    (void)rbe_user_error(0,
                         "Internal error: Cannot generate trail
                                        queues, cannot continue.");
    return -1;
  }

  TrailRestoresLeft = TrailRestoresTotal;

  CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: RunWorkItemRestores for %d trails.",
                         TrailRestoresTotal);
  }
  /*
   * This is the start up loop to get the initial work item
   * restores started.
   */

  QuitFlag = CancelRestoreTest();

  while((CountDrivesInUse < CountDrivesAvailable) &&
        (HighestActiveTrail < TrailRestoresTotal) &&
        (FALSE == QuitFlag))
  {
    int submitObjID = 0;
    int submitelementID = 0;

    HighestActiveTrail++;

    /*
     * Activate the Trail Queue.
                                 This allows the traill queues to be used to
     * determine the work item restores to run.
     */
    if(0 != ActivateTrailQueue(HighestActiveTrail,
                               1,
                               &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot activate trail
                       queues(1) for trailid %d, cannot continue.",
                           HighestActiveTrail);
      return -1;
    }
```

```
static int
DetermineGlobalDriveUse();

static int
test_fd_hup(int fd);

static int
FindTrailIDForWItem(int handle,
                    int *TrailID,
                    int *status);

static int
SendRunningWorkItemsQuit();

/* End Stubs */

static int
Select(int nfds,
       fd_set *readfds,
       fd_set *writefds,
       fd_set *exceptfds,
       struct timeval *timeout);


static int
HandleWorkItemRestoreResults(int FromFD,
                             int *TrailID,
                             wi_restore_results *results);
static int
RunWorkItemRestoresForTrail(const int TrailID,
                            const int CountDrivesAvailable,
                            boolean_ty (*CancelRestoreTest)(),
                            boolean_ty *QuitFlag,
                            int *CountDrivesInUse);

/*
 * RunWorkItemRestores()
 *
 *  Runs a set of work item restores.
 *
 * Args:
 *  SubmitObject
 *  CancelRestoreTest().
 *
 * Returns: int 0 for success.
 */
int
RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)(
                                                                      ))
{
  boolean_ty QuitFlag = FALSE;  /* Has the user requested a quit.*/
  boolean_ty SentQuit = FALSE;  /* Have we initiated the quit. */

  int TrailRestoresRunning = 0;
                            /* The number of trail restores running. */
  int TrailRestoresLeft;
                               /* The number of trail restores left. */
  int TrailRestoresTotal;
                              /* The number of trail restores total. */

  int CountDrivesAvailable;     /* The count of drives available. */
  int CountDrivesInUse = 0;     /* The count of drives in use. */

  int temp_status;
  int HighestActiveTrail = 0;
                           /* The trail queues are ordered from 1 to n. */
```

```
                                     /* This is the highest trail running
                                                                        */

  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: Running RunWorkItemRestores.");
  }
  /* GenerateTrailQueues()
   * Buckets the work items into trail queues.
                                            The trail queues are sorted
   * in the order which the restores should run.
   */

  if(0 != GenerateTrailQueues(SubmitObject,
                              &TrailRestoresTotal,
                              &temp_status))
  {
    (void)rbe_user_error(0,
                         "Internal error: Cannot generate trail
                                     queues, cannot continue.");
    return -1;
  }

  TrailRestoresLeft = TrailRestoresTotal;

  CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: RunWorkItemRestores for %d trails.",
                         TrailRestoresTotal);
  }
  /*
   * This is the start up loop to get the initial work item
   * restores started.
   */

  QuitFlag = CancelRestoreTest();

  while((CountDrivesInUse < CountDrivesAvailable) &&
        (HighestActiveTrail < TrailRestoresTotal) &&
        (FALSE == QuitFlag))
  {
    int submitObjID = 0;
    int submitelementID = 0;

    HighestActiveTrail++;

    /*
     * Activate the Trail Queue.
                              This allows the traill queues to be used to
     * determine the work item restores to run.
     */
    if(0 != ActivateTrailQueue(HighestActiveTrail,
                               1,
                               &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot activate trail
                   queues(1) for trailid %d, cannot continue.",
                           HighestActiveTrail);
      return -1;
    }
```

```
    /*
     * This sets the number of drives and media access concurrency for
                                                        trail restores.
     * i.e. The count of running work item restores for this trail.
                                                       Today this is one.
     */
    if(0 != SetTQDrivesAcquired(HighestActiveTrail, 1, &temp_status))
    {
      (void)rbe_user_error(0,
                          "Internal error: Cannot set drive acquired(
            1) for trailid %d, cannot continue.", HighestActiveTrail);
      return -1;
    }

    if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                     HighestActiveTrail,

                                                   CountDrivesAvailable,

                                                       CancelRestoreTest,
                                                           &QuitFlag,

                                                     &CountDrivesInUse)))
    {

      /* RunWorkItemRestoresForTrail does its own error logging. */
      return -1;
    }

    if(temp_status == 0)
    {
      (void)rbe_log_stats(0,
                          "Trail %d restore had no work item to run(
                                       1).", HighestActiveTrail);
      /* more work may be need to recover from this error condition.
                                                                    */
    }
    if(temp_status > 0)
    {
      TrailRestoresRunning++;
    }
  } /* End while() initial startup loop */

  while(1)
  {
    int HighestFd = 0;
    fd_set WorkItemFromFds;
    int retStatus;
    struct timeval timeout = {5, 0};

    if((QuitFlag) && (!SentQuit))
    {
      (void)rbe_log_stats(0,
                          "Restore was quit by user. Quitting restore,
                                       this could take a while.");

      SendRunningWorkItemsQuit();
      SentQuit = TRUE;
    }

    if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
    {
      (void)rbe_user_error(0,
                          "Internal error: Cannot get auxproc result
                                       fds, cannot continue.");
```

```
      return -1;
    }
#if 0
    if(debugmode)
    {
      DebugLogFds("The file descriptors to wait on are ",
                  &WorkItemFromFds);
    }
#endif
    if(0 > (retStatus = Select(HighestFd + 1,
                               &WorkItemFromFds,
                               NULL, NULL,
                               &timeout)))
    { /* error */
      (void)rbe_user_error(RBRECOVER_MKERR(errno),
                           "Internal error: Cannot get auxproc result
                                         fds, cannot continue.");

      return -1;
    }

    else if (0 == retStatus)
    { /* timed out */

      QuitFlag = CancelRestoreTest();

    }

    else
    { /* Available fds */

      int ReadyFds = retStatus;
      int FoundFds = 0;
      int index;

      if(debugmode)
      {
        DebugLogFds("The file descriptors ready to read are ",
                    &WorkItemFromFds);
      }

      /*
       * If there are available fds then we may want to
       * schedule the next work item restore. We should
       * check if the user initiated a quit.
       */
      QuitFlag = CancelRestoreTest();

      for(index = 0;
          (index < (HighestFd + 1)) && (FoundFds < ReadyFds);
          index++)
      {
        int StartWorkItemforTrail = 0;
        if(FD_ISSET(index, &WorkItemFromFds))
        {
          int TrailID;
          int TrailAcquired;
          wi_restore_results results;
          FoundFds++;

          memset(&results, 0, sizeof(wi_restore_results));

          if(0 != HandleWorkItemRestoreResults(index,
                                               &TrailID,
                                               &results))
          {
                /* HandleWorkItemRestoreResults will do its own logging!
                                                                        */
                return -1;
            }
            /*
             * This is where we may want to retry the Work Item
             * Based on if it passes or fails
             */

          CountDrivesInUse--;


          if (0 > (StartWorkItemforTrail =
                   RunWorkItemRestoresForTrail(TrailID,
                                               CountDrivesAvailable,
                                               CancelRestoreTest,
                                               &QuitFlag,
                                               &CountDrivesInUse)))
          {
            /* RunWorkItemRestoresForTrail does its own logging. */
            return -1;
          }

          else if (StartWorkItemforTrail == 0)
               /* 0 work items started aboved,
                * lets check to see if this is the last work item
                                                               left for
                * this trail
                */
          {
            int wiCount;

            if(0 != GetRunningWI(TrailID, &wiCount, &temp_status))
            {
              (void)rbe_user_error(0,
                                   "Internal error: Cannot determine
          number of running work items for trail, cannot continue.");

              return -1;
            }
            if(debugmode)
            {
              (void)rbe_user_error(0,
                                   "DEBUG: RunWorkItemRestores no
                              more work items left for trailid %d ,
                            but %d wiCount workitem still running.",
                                      TrailID, wiCount);
            }
            /* Testing for No work items left running or started
             * For this trail.
             */

            if((0 == wiCount) && (0 == StartWorkItemforTrail))
            {

              TrailRestoresRunning--;
              TrailRestoresLeft--;

              if(0 != DeactivateTrailQueue(TrailID, &temp_status))
              {
                (void)rbe_user_error(0,
                                     "Internal error: Cannot
    deactivate trail queue for trailid %d, cannot continue.", TrailID);

                return -1;
              }
```

```
              /* This test is to determine,
                                            if a trail restore finished,
               * there may be another not yet started trail,
                                                             If we have
               * drive available the next trail restore will be
                                                                started.
               */
              if ((0 != TrailRestoresLeft) &&
                  (HighestActiveTrail < TrailRestoresTotal) &&
                  (CountDrivesInUse < CountDrivesAvailable))
              {

                HighestActiveTrail++;

                if(0 != ActivateTrailQueue(HighestActiveTrail,
                                           1,
                                           &temp_status))
                {
                  (void)rbe_user_error(0,
                                       "Internal error: Cannot
      activate trail queue(2) for trailid %d, cannot continue.",
                                       HighestActiveTrail);
                  return -1;
                }

                if(0 != SetTQDrivesAcquired(
                           HighestActiveTrail, 1, &temp_status))
                {
                  (void)rbe_user_error(0,
                                       "Internal error: Cannot set
            drive acquired(2) for trailid %d, cannot continue.",
                                       HighestActiveTrail);
                  return -1;
                }

                if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                              HighestActiveTrail,

                                            CountDrivesAvailable,

                                               CancelRestoreTest,

                                                       &QuitFlag,

                                              &CountDrivesInUse)))
                {
                  /* RunWorkItemRestoresForTrail does its own logging.
                                                                     */
                  return -1;
                }
                if(temp_status == 0)
                {
                  /* If this Trail Had no work items we
                   * Should attempt to run the next trails
                   * work items here. This would be an internal
                   * error if a trail queue had no work item
                   * restores.
                   */
                  (void)rbe_log_stats(0,
                                      "Internal error: Trail %d
      restore had no work item to run(1).", HighestActiveTrail);
                  return -1;
                }
                if(temp_status > 0)
                {
                        /* IF at least on work item was started for this
                                                                    trail,
                         * then we have started a new trail.
                         */
                        TrailRestoresRunning++;
                     }
                  }

               }
            }
         }
      } /* end for() */

    } /* else Available fds */

    /*
     * Terminate the loop if either
     * ----------------------------
     * 1) Sent the work items the quit AND
     *    No Trail restores a running.
     * OR
     * 2) No more Trail restores are left.
     */

    if(((0 == TrailRestoresRunning) && (SentQuit)) ||
       (0 == TrailRestoresLeft))
    {
      break;
    }

  }  /* end while(1) */
  if((0 == TrailRestoresRunning) && (SentQuit))
  {
    (void)rbe_log_stats(0,
                        "Restore was quit by user.
                                        Work item restore quit.");

  }
  return 0;
}
```

```
/* Functions needed
SendRunningWorkItemsQuit();
InterpretWorkItemRestoreResults();
*/

static int
Select(int nfds,
       fd_set *readfds,
       fd_set *writefds,
       fd_set *exceptfds,
       struct timeval *timeout)
{
  int retSelect;

  do
  {
    retSelect = select(nfds,
                       readfds,
                       writefds,
                       exceptfds,
                       timeout);

  } while ((-1 == retSelect) && (EINTR == errno));

  return retSelect;
}
```

```
eperrno
InitiateWorkItemRestore(const int SubmitObjID,
                        const int SubmitElemID)
{
  struct auxproc AuxprocVitals;
  eerrno_ty StartupAPResults = EXIT_FAILURE;
  time_t StartTime;
  int TempStatus;
  char junk_executable[1024];
  char **junk_argv;
  char **AP_env = NULL;
  int SOstatus;
  char clientName[256] = "";
  int clientPort;
  int status;
  time_t EndTime;
  /* Lets see if there are any environment variables to set.
   *  The restore of the output variables are ingnored.
   */
  if(E_SUCCESS != GetSOExecutionPhase(SubmitObjID,
                                      junk_executable, 1024,
                                      &junk_argv,
                                      &AP_env,
                                      &SOstatus))
  {
    (void)rbe_user_error(0,
                          "Internal Error: Could not get environment
                                                          variables.");
    return -1;
  }

  if (E_SUCCESS == GetSERcmdConnect(SubmitObjID,
                                    SubmitElemID,
                                    clientName, 256,
                                    & clientPort,
                                    &SOstatus))
  {
   StartupAPResults = StartupAuxprocess(0 /* XXX */,
                                        &AuxprocVitals,
                                        AP_env,
                                        clientName,
                                        clientPort);
  }
  else
  {
    (void)rbe_user_error(0,
                   "Internal Error: Could not get Remote Client name
                                                                  &"
                   "port to connect.");
    return -1;
  }

  if(E_SUCCESS != StartupAPResults)
  {
    /* StartupAuxprocess does its own logging. */
    return -1;
  }

  /*
   * We need to close the bulk fd. This file descriptor
   * is not being used any more. If we do not close it
   * here we will have a file descriptor leak because
   * we won't be able to determine what it was when the
   * work item completes.
```

```
   */

  close(AuxprocVitals.xp_fd_bulk_to_x);

  time(&StartTime);

  if(0 != newHandleSet(AuxprocVitals.xp_fd_to_x,
                       AuxprocVitals.xp_fd_from_x,
                       AuxprocVitals.xp_fd_prog_from_x,
                       SubmitObjID,
                       SubmitElemID,
                       AuxprocVitals.xp_pid,
                       StartTime,
                       &TempStatus))
  {
    (void)rbe_user_error(
                  0, "Internal Error: Could not register handle set.");
    return -1;
  }


  if(0 > StartWorkItemRestore(rcp,
                              &AuxprocVitals,
                              SubmitObjID,
                              SubmitElemID))
  {
    /*
     * StartWorkItemRestore does logging if initialization fails
     */
     (void)rbe_user_error(
                 0, "Error in StartWorkItemRestore  SubmitObjID %d,"
                        "SubmitElemID %d ",SubmitObjID,
                                                    SubmitElemID);
     /*
      * the following code kills auxproc when recx or xcpio do not
                                                               start
      * we do not want an auproc sitting around.
      * if errors occur in deleteHandleSet or KillWorkItemRestore the
                                                         messages are
      * logged in those calls, plus,
                       we already know there was an error and that
      * is why we are doing this right now.
      */
     time(&EndTime);

     deleteHandleSet(
  AuxprocVitals.xp_fd_from_x, EndTime, EP_RB_RECOVER_ALLFAIL, &status);

     KillWorkItemRestore(
                  AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_x);

     return -1;
  }

  return 0;

} /* InitiateWorkItemRestore() */
```

```
/*
 *       interperate return.
 *       Drain progress.
 *       Send final Progress for work item.
 *       Delete the handle set.
 */

static int
HandleWorkItemRestoreResults(int FromFD,
                             int *TrailID,
                             wi_restore_results *results)
{
  int ret = 0;
  int retries = 0;
  int GetAuxprocResultsStatus;
  int TempStatus;
  int DrainResult;
  int DrainedFD;
  int wiCount;
  int AuxProcPid;
  int ToFD, getFromFD, ProgressFD;
  time_t EndTime;
  unsigned long jobstat;
  int timeout = 3; /* Lets try 3 seconds */
  boolean_ty fromFDHangUp = FALSE;

  ToFD = getFromFD = ProgressFD = -1;

  while(! (fromFDHangUp))
  {
    GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);

    if(-1 == GetAuxprocResultsStatus)
    {
      /* GetAuxprocResults() does its own logging */

     (void)rbe_user_error(0," Error in GetAuxprocResults");
      return -1;
    }
    if(0 == GetAuxprocResultsStatus)
    {
        if(test_fd_hup(FromFD) == 1)
        {
           fromFDHangUp = TRUE;
        }
    }

    /* The remote result are not always going to
     * be set. For example if the remote command
     * is not started correctly.
     */

    if(results->local_exit_set == TRUE)
    {
      break;
    }
    else
    {
      sleep(1);
      test_fd(FromFD);
      continue;
    }
  }
```

```
  time(&EndTime);

  if(0 != PushDrainRequest(FromFD, &TempStatus))
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not push drain
                                          request, cannot continue.");

    return -1;
  }
  /* Lets give the progress thread a chance to drain keeping busy in
   * the meanwhile.
   */

  if(0 != FindTrailQueueOfWI(FromFD, TrailID, &TempStatus))
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not find trail id for
                                   finished work item, cannot continue.");

    return -1;
  }

  if(0 != DecrementRunningWI(*TrailID, &wiCount, &TempStatus))
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not decrement running
                                work items for trail, cannot continue.");

    return -1;
  }

  if(0 != getPID(FromFD, &AuxProcPid, &TempStatus))
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not get auxproc pid
                                        for work item, cannot continue.");

    return -1;
  }

  if(0 != getHandleSet(
                 FromFD, &ToFD, &getFromFD, &ProgressFD, &TempStatus))
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not get auxproc file
                           descriptors for work item, cannot continue.");

    return -1;
  }
  if(FromFD != getFromFD)
  {
    (void)rbe_user_error(0,
                              "Internal error:  mismatch on from file
                           descriptors for work item, cannot continue.");
    return -1;
  }

  while (0 != (ret = PopDrainResult(timeout,
                                    &DrainResult,
                                    &DrainedFD,
                                    &TempStatus)) && retries < 3)
     retries++;

  if (ret != 0)
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not pop drain results,
                                                 cannot continue.");

    return -1;
  }

  /*
   *       Send final Progress for work item. XXX
   */

  /* Translate the local and remote error statuses
   * to an eperrno value:
   */

  if(0 != results->local_exit_status)     /* use local error, if any */
    switch (results->local_exit_status)
    {
    case XG_EXIT_ALLFAIL:
      jobstat = EP_RB_RECOVER_ALLFAIL;
      break;
    case XG_EXIT_MANYFAIL:
      jobstat = EP_RB_RECOVER_MANYFAIL;
      break;
    case XG_EXIT_FEWFAIL:
      jobstat = EP_RB_RECOVER_FEWFAIL;
      break;
    case SPEXIT_REMOTE_STDERR_PROTOCOL:
    case SPEXIT_REMOTE_STDERR_FAIL:
      jobstat = EP_RB_RECOVER_CLIENT_STDERR_FAIL;
      break;
    case XG_EXIT_STOPPED:        /* treat like signal */
    default:      /* check for signal termination vs all generic
                                                      failures */
      if ( XG_EXIT_SIGBASE < results->local_exit_status
        || XG_EXIT_STOPPED == results->local_exit_status )
      { /* killed by signal or stopped: separate error for sigpipe */
        if (XG_EXIT_SIGBASE + SIGPIPE == results->local_exit_status)
          jobstat = EP_RB_RECOVER_SERVER_SIGPIPE;
        else
          jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
      }
      else
      { /* generic server failure, unless client failed too */
        jobstat = EP_RB_RECOVER_SERVERFAIL;
        if (0 != results->remote_exit_status)
          jobstat = EP_RB_RECOVER_BOTHFAIL;
      }
    }
  else if(0 != results->remote_exit_status)
    jobstat = EP_RB_RECOVER_CLIENTFAIL;
  else
    jobstat = E_SUCCESS;

  if( (0 != results->remote_exit_status) ||
      (0 != results->local_exit_status ) )
  {
   int  status=0;
   int  rc=0;
   char *templateName=NULL;
   char *winame=NULL;
   char *trailsetName=NULL;

   rc = getHandleSetInformation(FromFD,
                                &templateName,
                                &winame,
```

```
                                &trailsetName,
                                &status);
   rbe_log_stats(0, "Restore Failure of "\
                "top level object: %s, template %s.",
                STR_SURE(winame),
                STR_SURE(templateName));
   free(templateName);
   free(winame);
   free(trailsetName);
  }

  if( 0 != deleteHandleSet(FromFD, EndTime, jobstat, &TempStatus))
  {
      (void)rbe_user_error(0,
                              "Internal error: Could not delete Handle
                                                 Set, cannot continue.");

    return -1;
  }

  if(0 != KillWorkItemRestore(AuxProcPid,
                              -1 /* Hack this arg is not needed yet
                                                     cmd_to */))
  {
    (void)rbe_user_error(0,
                              "Internal error: Could not kill finished
                                          auxproc, cannot continue.");

    return -1;
  }

  close(ToFD);
  close(FromFD);
  close(ProgressFD);

  if(debugmode)
  {
    (void)rbe_user_error(0,
                              "DEBUG: HandleWorkItemRestoreResults Auxproc(
           PID %d) just finished for trailid %d work items left = %d.",
                              AuxProcPid,
                              *TrailID,
                              wiCount);

    (void)rbe_user_error(0,
                              "DEBUG: HandleWorkItemRestoreResults Auxproc(
            PID %d) results are local: %d: setP:%s remote: %d set:%s.",
                              AuxProcPid,
                              results -> local_exit_status,
                              (
                            results -> local_exit_set) ? "TRUE": "FALSE" ,
                              results -> remote_exit_status,
                              (
                           results -> remote_exit_set) ? "TRUE": "FALSE");


  }
  return 0;
} /* End HandleWorkItemRestoreResults() */
```

```c
/*
 * RunWorkItemRestoresForTrail()
 *
 * Description
 *  This function starts all the work item for the
 *  trail. For no this is set to one but concurrency
 *  will can be supportted.
 *
 * Args:
 *  (I) TrailID -- The id for this trail.
 *  (I) CountDrivesAvailable -- the total drives available to restore.
 *  (O) QuitFlag -- indicatate whether the user has quit the restore.
 *  (O) CountDrivesInUse -- The count of trails in use by restore.
 *
 * Return int
 *  if -1 then an error has occurred.
 *  if 0 or greater then the number of trail restores started will be
 *  returned.
 *
 */
static int
RunWorkItemRestoresForTrail(const int TrailID,
                            const int CountDrivesAvailable,
                            boolean_ty (*CancelRestoreTest)(),
                            boolean_ty *QuitFlag,
                            int *CountDrivesInUse)
{
  int DriveAcquiredForTrail;
  int DriveConcurencyForTrail;
  int submitObjID;
  int submitelementID;
  int popResults = 0;
  int temp_status;
  int CountOfWorkItemRestoreStarted = 0;
  int wiCount;

  while(1)
  {


    (*CountDrivesInUse)++;

    if((0 != (popResults = PopWIFromTrailQueue(TrailID,
                                               &submitObjID,
                                               &submitelementID,
                                               &temp_status))) &&
       (SCHED_NO_MORE_JOBS != temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot pop work item off
                                  trail queue, cannot continue.");
      return -1;
    }

    if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
    {
      return CountOfWorkItemRestoreStarted;
    }

    temp_status = InitiateWorkItemRestore(
                                   submitObjID, submitelementID);

    if(temp_status != 0)
```

```c
    {
      /* InitiateWorkItemRestore() does its own logging */
      (void)rbe_user_error(0, "Error in InitiateWorkItemRestore,"
           "submitObjID  %d, submitelementID %d", submitObjID,
                                                   submitelementID);
      return -1;
    }

    if(0 != IncrementRunningWI(TrailID, &wiCount, &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Could not increment
                    running work items for trail, cannot continue.");
      return -1;
    }

    CountOfWorkItemRestoreStarted++;

    if(0 != GetTQDrivesAcquired(TrailID,
                                &DriveAcquiredForTrail,
                                &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot get drives
                                  acquired, cannot continue.");

      return -1;
    }

    if(0 != GetTQDriveConcurrency(TrailID,
                                  &DriveConcurencyForTrail,
                                  &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot get drive
                                  concurrency, cannot continue.");
      return -1;
    }

    *QuitFlag = CancelRestoreTest();

    if((DriveAcquiredForTrail < DriveConcurencyForTrail) &&
       ((*CountDrivesInUse) < CountDrivesAvailable) &&
       (FALSE == *QuitFlag))
    {
      continue;
    }
    else
    {
      break;
    }
  }
  return CountOfWorkItemRestoreStarted;
} /* RunWorkItemRestoresForTrail() */
```

```
/* Stub */
static int DetermineGlobalDriveUse()
{
/* Limiting to MAXINT === not limiting... Need resource management
 * to do this properly.
                    NOTE: This should now work like eb_dc_restore does.
 */
  return MAXINT;
}
```

```
static int
SendRunningWorkItemsQuit()
{
  int *APlist;
  int count;
  int status;
  int index;

  if(0 != getPIDList(&count, &APlist, &status))
  {
    (void)rbe_user_error(0,
                         "Internal error: Cannot get auxproc pid list,
                                                     cannot continue.");
    return -1;
  }

  for(index = 0; index < count; index++)
  {
    QuitWorkItemRestore(APlist[index]);
  }
  return 0;
}
```

```
/*
 * Stub this out for now.
 */
static int
InterpretWorkItemRestoreResults(wi_restore_results *results)
{
  return 0;
}
```

```
static void
DebugLogFds(char *error_msg,
             fd_set *fds)
{
  int index, fd_count = 0;
  char buffer[4096];
  char *bufptr = (char*)buffer;


  for(index=0;
      index < 1024;
      index++)
  {
    if( FD_ISSET(index, fds))
    {
      int size = 0;
      size = sprintf(bufptr,"%d,",index);
      bufptr += size;
      fd_count++;
    }
  }
  rbe_log_stats(0,"%s fd_count:: %d :: {%s}\n",
                error_msg, fd_count, buffer);
}
```

```
static int
test_fd(int fd)
{
  fd_set read_fd;
  int ret_select;
  struct timeval timeout = {0, 0};

  FD_ZERO(&read_fd);

  FD_SET(fd, &read_fd);

  do
  {
    ret_select = select(fd + 1, &read_fd, NULL, NULL,  &timeout);

  } while((-1 == ret_select) && (EINTR == errno));

 return ret_select;

}
```

```
/*
 * test_fd_hup()
 *
 * Description: Test the supplied file descriptor to see if
 *   it has had the hang up condition.
 *
 * Args:
 *  Input fd -- the file descriptor to check for the hang up condition.
 *
 * Returns:
 *  1 for HUP event received on fd.
 *  0 No HUP event received on fd.
 *  -1 errno set.
 *
 */
static int
test_fd_hup(int fd)
{
  struct pollfd fds;
  int ret_poll;

  if(fd < 0)
  {
    errno = EINVAL;
    return -1;
  }

  fds.fd = fd;
  fds.events = POLLIN;
  fds.revents = 0;  /* initialize */

  do
  {
    ret_poll =  poll(&fds, 1, 0);

  } while((-1 == ret_poll) && (EINVAL == errno));

  if(-1 == ret_poll)
  {
    return -1;
  }

  if(POLLHUP & fds.revents)
  {
    return 1;
  }
  else
  {
    return 0;
  }

} /* end test_fd_hup() */
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMRESchedApi.cc
 *
 *
 * Mission Statement:  file that contains an API to manage the order
                                                            in which
 *                     restores occur
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 *                     A few calls to manage the order in which work
                                                               items
 *                     are run.
                            Currently things are ordered by where they
 *                     appear in the Submit list but this will need to
 *                     change in the future.
 */

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMRESchedApi.cc,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>

#include <restore/REprogmsg.h>
#include <restore/dispatch_daemon.h>

#include <restore/EDMRESubmitApi.h>
#include <EDMREHandleMgrApi.h>
#include <EDMREScheduledWI.h>
#include <EDMRETrailList.h>
#include <EDMRESchedApi.h>

static unsigned int  numberOfQueues = 0;
static RWBinaryTree trailLists;
static pthread_mutex_t    G_scheduleMtx = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    char        templatename[TEMPLNAME_SIZE];
    boolean_ty alternate;
    int         trailnum;
} findArg;

/*********************************************************************
                                                            ********
```

```
**
** Routine:  LockScheduleMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the mutex for the handle tree object
**
*********************************************************************
                                                            ********
*/

static void
LockScheduleMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_scheduleMtx, NULL);
    }

    pthread_mutex_lock(&G_scheduleMtx);
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  UnlockScheduleMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for the handle tree object
**
*********************************************************************
                                                            ********
*/

static void
UnlockScheduleMutex()
{
    pthread_mutex_unlock(&G_scheduleMtx);
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  LookupSchedWI
**
** Inputs:   int ID - trail object ID associated with element
**           int jobID - wi job ID associated with element
**
** Outputs:  int *status - status of the function if an error occured
**           EDMRESchedWI **wi - place to put the pointer to the
                                                             element
**
** Return Codes:
**           int - 0 for success or non-zero for failure
**
** Purpose:  Finds a scheduled work item based on the trail ID and the
**           work item ID.
**
*********************************************************************
                                                            ********
*/

static int
LookupSchedWI(int ID, int jobID, EDMREScheduledWI **wi,
                    int *status)
{
    EDMRETrailList *trl;
    EDMRETrailList *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (wi == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    trl = new EDMRETrailList();

    if (trl == NULL)
    {
        *status = SCHED_NO_MEMORY;
        return -1;
    }

    trl -> setTrailQID(ID);

    LockScheduleMutex();

    ret = (EDMRETrailList *) trailLists.find(trl);

    delete trl;

    if (ret == NULL)
    {
        *status = SCHED_TRAIL_LOOKUP_FAILED;
        UnlockScheduleMutex();
        return -1;
    }

    *wi = ret -> getScheduledWI(jobID);
    UnlockScheduleMutex();

    if (*wi == NULL)
    {
        *status = SCHED_JOB_LOOKUP_FAILED;
        return -1;
    }

    return 0;
}
```

```
/*************************************************************************
                                                                 ********
**
** Routine:  LookupTrailObject
**
** Inputs:   int ID - trail object ID
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 for success or non-zero for failure
**
** Purpose:  Finds a trail object based on the trail ID.
**
*************************************************************************
                                                                 ********
*/

static int
LookupTrailObject(int ID, EDMRETrailList **trl, int *status)
{
    EDMRETrailList *tmptrl;
    EDMRETrailList *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (trl == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    tmptrl = new EDMRETrailList();

    if (tmptrl == NULL)
    {
        *status = SCHED_NO_MEMORY;
        return -1;
    }

    tmptrl -> setTrailQID(ID);

    LockScheduleMutex();

    ret = (EDMRETrailList *) trailLists.find(tmptrl);

    UnlockScheduleMutex();

    delete tmptrl;

    if (ret == NULL)
    {
        *status = SCHED_TRAIL_LOOKUP_FAILED;
        return -1;
    }

    *trl = ret;

    return 0;
}
```

```
/*********************************************************************
********
**
** Routine:  NewTrailObject
**
** Inputs:   NONE
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - ID of the new trail Object
**
** Purpose:  Creates a new trail object and inserts it in the trail
                                                                list.
**
*********************************************************************
********
*/

int
NewTrailObject(int *status)
{
    EDMRETrailList *tl;
    EDMRETrailList *ret;

    if (status == NULL)
    {
        return 0;
    }

    tl = new EDMRETrailList();

    if (tl == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return 0;
    }

    tl -> setTrailQID(++numberOfQueues);

    LockScheduleMutex();

    ret = (EDMRETrailList *) trailLists.insert(tl);

    UnlockScheduleMutex();

    if (ret == NULL)
    {
        *status = SCHED_TRAIL_INSERT_FAILED;
        delete tl;
        return 0;
    }

    return numberOfQueues;
}
```

```
/*********************************************************************
********
**
** Routine:  NewSchedWI
**
** Inputs:   int ID - trail ID associated with new element
**           int submitID - submit ID associated with new element
**           int elementID - submit element ID associated with new
                                                              element
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - ID of the new sched WI element
**
** Purpose:  Creates a new scheduled work item element and inserts it
                                                               in the
**           trail Q tree.
**
*********************************************************************
********
*/

int
NewSchedWI(int ID, int submitID, int elementID, int *status)
{
    EDMRETrailList *trl;
    EDMRETrailList *ret;
    EDMREScheduledWI *wi;
    int    winum = 0;

    if (status == NULL)
    {
        return 0;
    }

    trl = new EDMRETrailList();

    if (trl == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return 0;
    }

    trl -> setTrailQID(ID);

    LockScheduleMutex();

    ret = (EDMRETrailList *) trailLists.find(trl);

    delete trl;

    if (ret == NULL)
    {
        *status = SCHED_TRAIL_LOOKUP_FAILED;
        UnlockScheduleMutex();
        return 0;
    }

    winum = ret -> newScheduledWI();

    if (winum <= 0)
    {
        *status = SCHED_NEW_JOB;
        UnlockScheduleMutex();
        return 0;
    }

    wi = ret -> getScheduledWI(winum);

    UnlockScheduleMutex();

    if (wi == NULL)
    {
        *status = SCHED_JOB_LOOKUP_FAILED;
        return 0;
    }

    wi -> setSubmitObjectID(submitID);
    wi -> setSubmitElementID(elementID);

    return winum;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  findTrail
**
** Inputs:   TrailList * - TrailList to check against
**
** Outputs:  findArg * - structure containing SubmitElement and status
**
** Return Codes:
**           None
**
** Purpose:  Sets the status to non-zero if a match is found.
**
*********************************************************************
                                                            ********
*/

static void
findTrail(IN RWCollectable* c, IN void *f)
{
    EDMRETrailList *trl = (EDMRETrailList *) c;
    findArg        *fa = (findArg *) f;
    char           trltempl[TEMPLNAME_SIZE];

    trl -> getTemplateName(trltempl, sizeof(trltempl));

    if (strcmp(fa->templatename, trltempl) == 0 &&
        fa -> alternate == trl -> isAlternateTrailset())
    {
        fa -> trailnum = trl -> getTrailQID();
    }
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  GenerateTrailQueues
**
** Inputs:   int ID - submit ID
**
** Outputs:  int *status - status of the function if an error occured
**           int *trailcount - place to put the trail queue count
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Generate the schedule ordered by trail.
**
**********************************************************************
                                                               ********
*/

int
GenerateTrailQueues(int ID, int *trailcount, int *status)
{
    EDMRESubmitObj *so;
    EDMRESubmitElement *se;
    EDMRETrailList *trl;
    int ret;
    int i;
    int numElements;
    findArg      fa;

    if (status == NULL)
    {
        return -1;
    }

    if (trailcount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupSubmitObject(ID, &so, status);

    if (ret != 0)
    {
        return ret;
    }

    numElements = so -> getWICount();

    for (i = 1; i < numElements + 1; i++)
    {
        se = so -> getSubmitElement(i);

        if (se == NULL)
        {
            *status = SCHED_SE_LOOKUP_FAILURE;
            break;
        }

        fa.alternate = se -> IsAlternateTrailset();
        se -> getTemplate(fa.templatename, TEMPLNAME_SIZE);
        fa.trailnum = 0;

        LockScheduleMutex();
```

```
        trailLists.apply(findTrail, &fa);

        UnlockScheduleMutex();

        if (fa.trailnum == 0)
        {
            char  setempl[TEMPLNAME_SIZE];

            fa.trailnum = NewTrailObject(status);

            if (fa.trailnum == 0)
            {
                break;
            }

            ret = LookupTrailObject(fa.trailnum, &trl, status);

            if (ret != 0)
            {
                break;
            }

            trl -> setSubmitID(ID);
            trl -> setAlternateTrailset(se -> IsAlternateTrailset());

            se -> getTemplate(setempl, sizeof(setempl));
            trl -> setTemplateName(setempl);
        }

        ret = NewSchedWI(fa.trailnum, ID, i, status);

        if (ret <= 0)
        {
            break;
        }
    }

    *trailcount = numberOfQueues;

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  PurgeTrailQueue
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Purge all lists.
**
************************************************************************
                                                                ********
*/

void
PurgeTrailQueue()
{
    LockScheduleMutex();

    trailLists.clearAndDestroy();

    UnlockScheduleMutex();

    numberOfQueues = 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  ActivateTrailQueue
**
** Inputs:   int ID - trail ID
**           int drivecount - number of drives to use
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Active the given trail for restore.
**
************************************************************************
                                                                ********
*/

int
ActivateTrailQueue(int ID, int drivecount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setTrailActive(TRUE);
    trl -> setMaxDrives(drivecount);

    return 0;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  DeactivateTrailQueue
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Deactivate the given trail for restore.
**
*********************************************************************
                                                            ********
*/

int
DeactivateTrailQueue(int ID, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setTrailActive(FALSE);

    return 0;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  SetTQDriveConcurrency
**
** Inputs:   int ID - trail ID
**           int drivecount - number of drives to use
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Set the drive concurrency for the given trail.
**
*********************************************************************
                                                            ********
*/

int
SetTQDriveConcurrency(int ID, int drivecount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setMaxDrives(drivecount);

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  GetTQDriveConcurrency
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *status - status of the function if an error occured
**           int *drivecount - place for the number of drives to use
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Retrieve the number of drives to use for the given trail
**           for restore.
**
************************************************************************
                                                                ********
*/

int
GetTQDriveConcurrency(int ID, int *drivecount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return  1;
    }

    if (drivecount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    *drivecount = trl -> getMaxDrives();

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  SetTQDrivesAcquired
**
** Inputs:   int ID - trail ID
**           int drivecount - the number of drives in use
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Sets the number of drives in use for the given trail
**           for restore.
**
************************************************************************
                                                                ********
*/

int
SetTQDrivesAcquired(int ID, int drivecount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setDrivesInUse(drivecount);

    return 0;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  GetTQDrivesAcquired
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *status - status of the function if an error occured
**           int *drivecount - a place to put the number of drives in
                                                                   use
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Gets the number of drives in use for the given trail
**           for restore.
**
**********************************************************************
                                                              ********
*/

int
GetTQDrivesAcquired(int ID, int *drivecount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (drivecount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    *drivecount = trl -> getDrivesInUse();

    return 0;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  IncrementRunningWI
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *wiCount - number of work items running after
                                                            increment.
**           int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Increment the running work items for the given trail.
**
**********************************************************************
                                                              ********
*/

int
IncrementRunningWI(int ID, int *wiCount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (wiCount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    *wiCount = trl -> IncrementRunningWIs();

    return 0;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  DecrementRunningWI
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *wiCount - number of work items running after
                                                          decrement.
**           int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Decrement the running work items for the given trail.
**
*********************************************************************
                                                              ********
*/

int
DecrementRunningWI(int ID, int *wiCount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (wiCount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    *wiCount = trl -> DecrementRunningWIs();

    return 0;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  SetRunningWI
**
** Inputs:   int ID - trail ID
**           int wiCount - number of work items to set.
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Set the running work items for the given trail.
**
*********************************************************************
                                                              ********
*/

int
SetRunningWI(int ID, int wiCount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setRunningWIs(wiCount);

    return 0;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  GetRunningWI
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *wiCount - number of work items running.
**           int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Get the running work items for the given trail.
**
*********************************************************************
                                                            ********
*/
int
GetRunningWI(int ID, int *wiCount, int *status)
{
    EDMRETrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (wiCount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    *wiCount = trl -> getRunningWIs();

    return 0;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  PopWIFromTrailQueue
**
** Inputs:   int ID - trail ID
**
** Outputs:  int *status - status of the function if an error occured
**           int *submitID - a place to put the submitID
**           int *elementID - a place to put the element ID
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Gets the submit ID and element ID of the next work item
                                                            to run.
**
*********************************************************************
                                                            ********
*/
int
PopWIFromTrailQueue(
                    int ID, int *submitID, int *elementID, int *status)
{
    EDMRETrailList *trl;
    EDMREScheduledWI *sw;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (submitID == NULL || elementID == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    if (trl -> isTrailActive() == FALSE)
    {
        *status = SCHED_TRAIL_NOT_ACTIVE;
        return -1;
    }

    sw = trl -> popScheduledWI();

    if (sw == NULL)
    {
        *status = SCHED_NO_MORE_JOBS;
        return -1;
    }

    *submitID = sw -> getSubmitObjectID();
    *elementID = sw -> getSubmitElementID();

    delete sw;

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  AddWIToTrailQueue
**
** Inputs:   int ID - trail ID
**           int submitID - the submitID of the work item
**           int elementID - the element ID of the work item
**
** Outputs:  int *status - status of the function if an error occured
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Add the work item described by the submit ID and element
                                                                   ID to
**           the specified trail queue.
**
************************************************************************
                                                                ********
*/

int
AddWIToTrailQueue(int ID, int submitID, int elementID, int *status)
{
    int ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = NewSchedWI(ID, submitID, elementID, status);

    if (ret <= 0)
    {
        return -1;
    }

    return 0;
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  FindTrailQueueOfWI
**
** Inputs:   int handle - handle to identify work item
**
** Outputs:  int *status - status of the function if an error occured
**           int *ID - a place to put the trail ID
**
** Return Codes:
**           int - 0 if success and non-zero otherwise
**
** Purpose:  Gets the trail ID of the work item.
**
***********************************************************************
                                                               ********
*/

int
FindTrailQueueOfWI(int handle, int *ID, int *status)
{
    EDMRESubmitElement *se;
    int                 soID,
                        seID;
    int                 ret;
    findArg             fa;

    if (status == NULL)
    {
        return -1;
    }

    if (ID == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = getSubmitIDs(handle, &soID, &seID, status);

    if (ret != 0)
        return ret;

    ret = LookupSubmitElement(soID, seID, &se, status);

    if (ret != 0)
        return ret;

    fa.alternate = se -> IsAlternateTrailset();
    se -> getTemplate(fa.templatename, TEMPLNAME_SIZE);
    fa.trailnum = 0;

    LockScheduleMutex();

    trailLists.apply(findTrail, &fa);

    UnlockScheduleMutex();

    if (fa.trailnum == 0)
        return -1;

    *ID = fa.trailnum;

    return 0;
```

```
}
```